Session Code: TLS320

tools & languages

Visual C# "Whidbey": Language Enhancements

Anders Hejlsberg
Distinguished Engineer
Microsoft Corporation
andersh@microsoft.com

PDC03

Make the connection

Microsoft

C# Language Enhancements

- Generics
- Anonymous methods
- Iterators
- Partial types
- Other enhancements

Generics

```
public class List<T>
{
    private T[] elements;
    private int count;

    public void Add(T element) {
        if (count == elements.Length)
Resize(count * 2);
        elements[count++] = element;
    }

    public T this[i
        get { returr
        set { eleme
    }

    public int Cou
        get { returr
    }
}
```

```
List<int> intList = new List<int>();

intList.Add(1);          // No boxing
intList.Add(2);          // No boxing
intList.Add("Three");    // Compile-time
error

int i = intList[0];      // No cast required
```

Generics

- Why generics?
  - Type checking, no boxing, no downcasts
  - Reduced code bloat (typed collections)
- How are C# generics implemented?
  - Instantiated at run-time, not compile-time
  - Checked at declaration, not instantiation
  - Work for both reference and value types
  - Complete run-time type information

Generics

- Type parameters can be applied to
  - Class, struct, interface, and delegate types

```
class Dictionary<K,V> {...}

struct HashBucket<K,V> {...}

interface IComparer<T> {...}

delegate
```

```
Dictionary<string,Customer>
customerLookupTable;

Dictionary<string,List<Order>>
orderLookupTable;

Dictionary<string,int> wordCount;
```

Generics

- Type parameters can be applied to
  - Class, struct, interface, and delegate types
  - Methods

```
class Utils
{
    public static
size) {
        return ne
    }

    public static void SortArray<T>(T[]
array) {
        ...
    }
}
```

```
string[] names =
Utils.CreateArray<string>(3);
names[0] = "Jones";
names[1] = "Anderson";
names[2] = "Williams";
Utils.SortArray(names);
```

Generics

- Type parameters can be applied to
  - Class, struct, interface, and delegate types
  - Methods
- Type parameters can have constraints

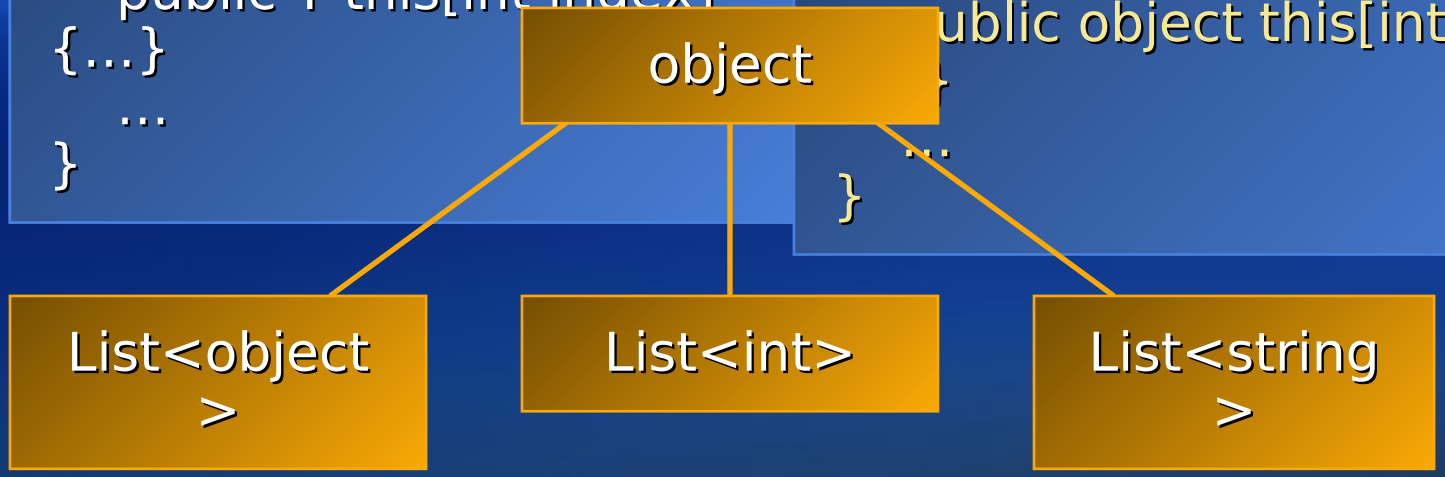
```
class Dictionary<K,V>: IDictionary<K,V>
    where K: IComparable<K>
    where V: IKeyProvider<K>, IPersistable, new()
{
    public void Add(K key, V value) {
        ...
    }
}
```

Generics

- Generics are “invariant”
- Interfaces can be used for type neutrality

```
public class List<T>: IList
{
    public void Add(T item)
{…}
    public T this[int index]
{…}
    …
}
```

```
public interface IList
{
    public void Add(object
item);
    public object this[int index]
    …
}
```



Generics

- T.default
- Null checks
- Type casts

```
void Foo<T>() {
   T x = null;              // Error
   T y = T.default;         // Ok
}
```

```
void Foo<T>(T x) {
   if (x == null) {
      throw new
FooException();
   }
   ...
```

```
void Foo<T>(T x) {
  int i = (int)x;                    //
Error
  int j = (int)(object)x;            //
Ok
}
```

Generics

- Collection classes
- Collection interfaces
- Collection base classes
- Utility classes
- Reflection

List<T>
Dictionary<K,V>
SortedDictionary<K,V>
Stack<T>
Qu

IList<T>
IDictionary<K,V>
ICollection<T>
IEnumerable<T>
IEnumerator<T>
IComparable<T>
IComparer<T>

Collection<T>
KeyedCollection<T>
ReadOnlyCollection<T>

Nullable<T>
EventHandler<T>
Comparer<T>

Generics

- System.Nullable<T>
  - Provides nullability for any type
  - Struct that combines a T and a bool
  - Conversions between T and Nullable<T>
  - Conversion from null literal to Nullable<T>

```
Nullable<int> x = 123;
Nullable<int> y = null;

int i = (int)x;
int j = x.Value;

if (x.HasValue)
Console.WriteLine(x.Value);
```

Anonymous Methods

```
class MyForm : Form
{
    ListBox listBox;
    TextBox textBox;
    Button addButton;

    public MyForm() {
        listBox = new ListBox(...);
        textBox = new TextBox(...);
        addButton = new Button(...);
        addButton.Click += delegate {
            listBox.Items.Add(textBox.Text);
        };
    }
}
```

Anonymous Methods

- Allows code block in place of delegate
- Delegate type automatically inferred
  - Code block can be parameterless
  - Or code block can have parameters
  - In either case, return types must match

```
button.Click += delegate
{ MessageBox.Show("Hello"); };
```

```
button.Click += delegate(object sender, EventArgs
e) {
   MessageBox.Show(((Button)sender).Text);
};
```

Anonymous Methods

- Co

```
public class Bank
{
    List<Account> GetLargeAccounts(double
minBalance) {
        Helper helper = new Helper();
        helper.minBalance = minBalance;
        return accounts.FindAll(helper.Matches);
    }

    internal class Helper
    {
        internal double minBalance;

        internal bool Matches(Account a) {
            return a.Balance >= minBalance;
        }
    }
}
```

Anonymous Methods

- Method group conversions
- Delegate type inferred when possible

```
using System;
using System.Threading;

class Program
{
   static void Work() {...}

   static void Main() {
      Thread t = new Thread(Work);
      t.Start();
   }
}
```

Generics Performance

demo

PDC03

Make the connection

Iterators

- foreach relies on “enumerator pattern”
  - GetEnumerator() method

```
foreach (object obj in list) {
   DoSomething(obj);
}
```

```
Enumerator e =
list.GetEnumerator();
while (e.MoveNext()) {
   object obj = e.Current;
   DoSomething(obj);
}
```

- foreach makes enumerating easy
  - But enumerators are hard to write!

Iterators

```
public class List
{
   internal object[]
   internal int coun

   public ListEnume
GetEnumerator() {
      return new Lis
   }
}
```

```
public class ListEnumerator : IEnumerator
{
   List list;
   int index;

   internal ListEnumerator(List list) {
      this.list = list;
      index = -1;
   }

   public bool MoveNext() {
      int i = index + 1;
      if (i >= list.count) return false;
      index = i;
      return true;
   }

   public object Current {
      get { return list.elements[index]; }
   }
}
```

Iterators

- Method that increr
  and returns a sequ
  - yield return and yie
  - Must return IEnume

```
public class List
{
   public IEnumerator
GetEnumerator() {
      for (int i = 0; i < count; i+
         yield return elements[i];
      }
   }
}
```

```
public IEnumerator GetEnumerator()
{
   return new __Enumerator(this);
}

private class __Enumerator:
IEnumerator
{
   object current;
   int state;

   public bool MoveNext() {
      switch (state) {
         case 0: ...
         case 1: ...
         case 2: ...
         ...
      }
   }

   public object Current {
      get { return current; }
   }
}
```

Iterators

```
public class List<T>
{
    public IEnumerator<T> GetEnumerator() {
        for (int i = 0; i < count; i++) yield return
elements[i];
    }

    public IEnumerable<T> Descending() {
        for (int i = count - 1; i >= 0; i--) yield return
elements[i];
    }

    public IEnumerable<T> Subrange(int index, int n) {
        fo
eleme
    }
}
```

```
List<Item> items = GetItemList();
foreach (Item x in items) {...}
foreach (Item x in items.Descending()) {...}
foreach (Item x in Items.Subrange(10, 20)) {...}
```

Partial Types

```
public partial class Customer
{
    private int id;
    private string na
    private string ad
    private List<Ord
}
```

```
public partial class
{
    public void Subn
    {
        orders.Add(or
    }

    public bool HasO
        return orders.
    }
}
```

```
public class Customer
{
    private int id;
    private string name;
    private string address;
    private List<Orders> orders;

    public void SubmitOrder(Order order)
    {
        orders.Add(order);
    }

    public bool HasOutstandingOrders()
    {
        return orders.Count > 0;
    }
}
```

Partial Types

demo

PDC03

Make the connection

Other Enhancements

- Static classes
  - Can contain only static members
  - Cannot be type of variable, parameter, etc.
  - System.Console, System.Environment, ...

```
public static class Math
{
   public static double Sin(double x)
{…}
   public static double Cos(double x)
{…}
   …
}
```

Other Enhancements

- Property accessor accessibility
  - Allows one accessor to be restricted further
  - Typically set {…} more restricted than

```
public class Customer
{
    private string id;

    public string CustomerId {
        get { return id; }
        internal set { id = value; }
    }
}
```

Other Enhancements

- Namespace alias qualifier
  - A::B looks up A only as namespace alias
  - global::X starts lookup in global namespace

```
using IO = System.IO;

class Program
{
   static void Main() {
      IO::Stream s = new
IO::File.OpenRead("foo.txt");
      global::System.Console.WriteLine("Hello");
   }
}
```

Other Enhancements

- Fixed size buffers
  - C style embedded arrays in unsafe code

```
public struct OFSTRUCT
{
    public byte cBytes;
    public byte fFixedDisk;
    public short nErrCode;
    private int Reserved;
    public fixed char szPathName[128];
}
```

Other Enhancements

- #pragma warning
  - Control individual warnings in blocks of code

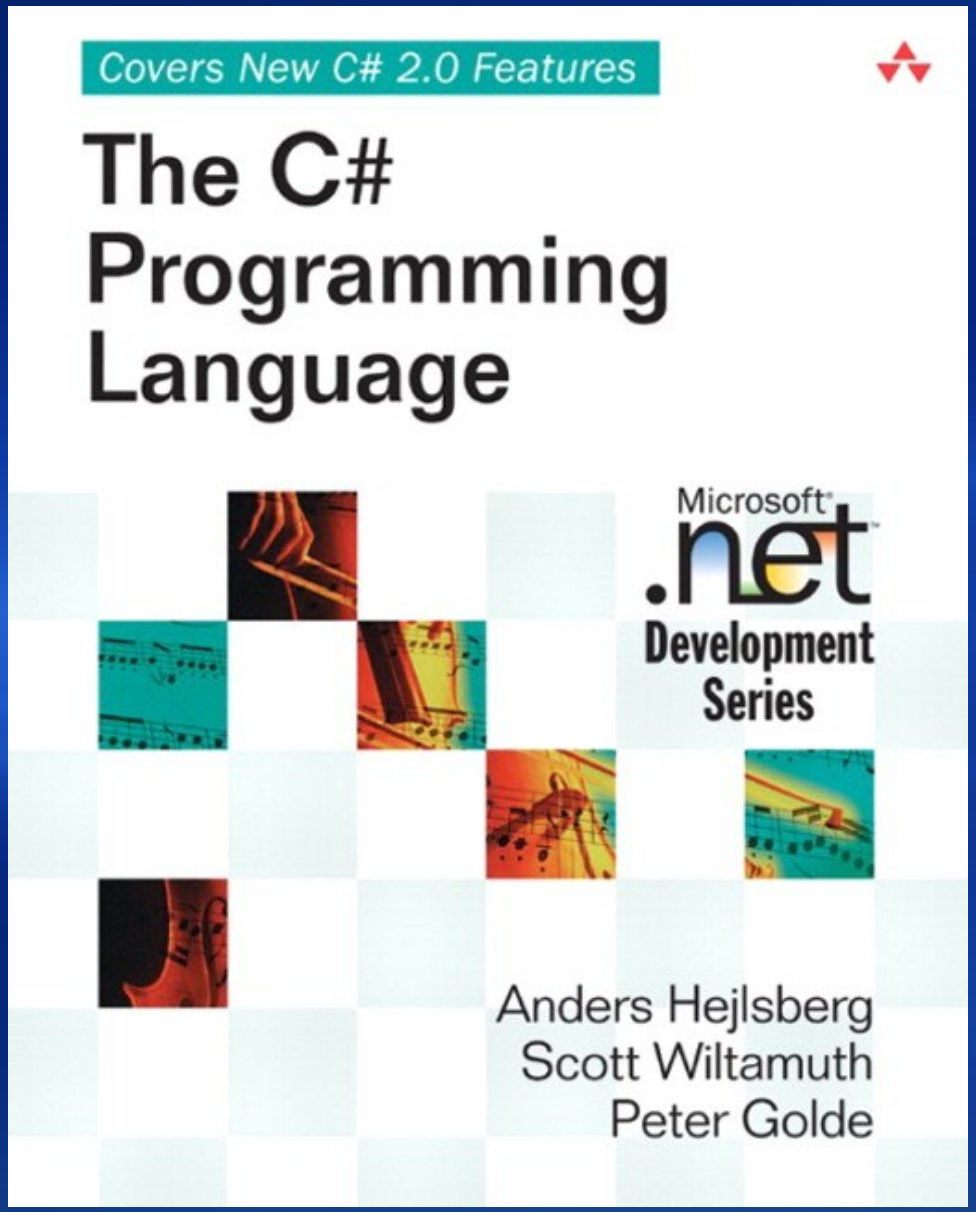
```
using System;

class Program
{
   [Obsolete]
   static void Foo() {}

   static void Main() {
#pragma warning disable 612
      Foo();
#pragma warning restore 612
   }
}
```

C# and CLI Standardization

- Work begun in September 2000
  - Intel, HP, IBM, Fujitsu, Plum Hall, and others
- ECMA standards ratified December 2001
- ISO standards published April 2003
- Several CLI and C# implementations
  - .NET Framework and Visual Studio .NET
  - “SSCLI” – Shared source on XP, FreeBSD, OS X
  - “Mono” – Open source on Linux



Book signing, Exhibition Hall Book Store, Wednesday, 1-2pm

Q & A

- C# Language home page
  - http://msdn.microsoft.com/vcsharp/language
- ECMA C# Standard
  - http://www.ecma-international.org/publications/standards/ecma-334.htm
- Panel: The Future of .NET Languages
  - PNL10, Thursday, 1:45pm – 3:15pm
- Tools Lounge (near Hall A)
  - Tuesday, 5:15pm – 6:30pm
  - Wednesday, 3:00pm – 5:00pm

Make the connection

Microsoft Professional Developers Conference 2003

October 26 - 30, 2003, Los Angeles, CA

Microsoft